Reprinted from MACHINE LANGUAGE PROGRAMMING FOR THE "8008" (and similar microcomputers).

Author: Nat Wadsworth

MACHINE LANGUAGE

Chapter I

THE '8008' CPU INSTRUCTION SET

The '8008' microprocessor has quite a comprehensive instruction set that consists of 48 basic instructions, which, when the possible permutations are considered, result in a total set of about 170 instructions.

The instruction set allows the user to direct the computer to perform operations with memory, with the seven basic registers in the CPU, and with INPUT and OUTPUT ports.

It should be pointed out that the seven basic registers in the CPU consist of one "accumulator," a register that can perform mathematical and logic operations, plus an additional six registers, which, while not having the full capability of the accumulator, can perform various useful operations. These operations include the ability to hold data, serve as an "operator" with the accumulator, and increment or decrement their contents. Two of these six registers have special significance because they may be used to serve as a "pointer" to locations in memory.

The seven CPU registers have arbitrarily been given symbols so that we may refer to them in an abbreviated language. The first register is designated by the symbol 'A' in the following discussion and will be referred to as the "accumulator" register. The next four registers will be referred to as the 'B,' 'C,' 'D' and 'E' registers. The remaining two special memory pointing registers shall be designated the 'H' (for the HIGH portion of a memory address) and the 'L' (for the LOW portion of a memory address) registers.

The CPU also has several "flip-flops" which shall be referred to as "FLAGS." The flip-flops are set as the result of certain operations and are important because they can be "tested" by many of the instructions with the instruction's meaning changing as a consequence of the particular status of a FLAG at the time the instruction is executed. There are four basic flags which will be referred to in this manual. They are defined as follows:

The 'C' flag refers to the carry bit status. The carry bit is a one unit register which changes state when the accumulator overflows or underflows. This bit can also be set to a known condition by certain types of instructions. This is important to remember when developing a program because quite often a program will have a long string of instructions which do not utilize the carry bit or care about its status, but which will be causing the carry bit to change its state from time-to-time. Thus, when one prepares to do a series of operations that will rely on the carry bit, one often desires to set the carry bit to a known state.

The 'Z' for zero flag refers to a one unit register that when desired will indicate whether the value of the accumulator is exactly equal to zero. In addition, immediately after an increment of decrement of the B, C, D, E, H or L registers, this flag will also indicate whether the increment or decrement caused that particular register to go to zero.

The 'S' for sign flag refers to a one unit register that indicates whether the value in the accumulator is a positive or negative value (based on two's complement nomenclature). Essentially, this flag monitors the most significant bit in the accumulator and is "set" when it is a one.

The 'P' flag refers to the last flag in the group which is for indicating when the accumulator contains a value which has even parity. Parity is useful for a number of reasons and is usually used in conjunction with testing for error conditions on words of data especially when transferring data to and from external devices. Even parity occurs when the number of bits that are a logic one in the accumulator is an even value. Zero is considered an even value for this purpose. Since there are eight bits in the accumulator, even parity will occur when zero, two, four or six bits are in the logic one condition regardless of what order they may appear in within the register.

It is important to note that the Z, S, and P flags (as well as the previously mentioned C flag) can all be set to known states by certain instructions. It is also important to note that some instructions do not result in the flags being set so that if the programmer desires to have the program make decisions based on the status of flags, the programmer should ensure that the proper instruction, or sequence of instructions is utilized. It is particularly important to note that load register instructions do not by themselves set the flags. Since it is often desirable to obtain a data word (that is, load it into the accumulator) and test its status for such parameters as whether or not the value is zero, or a negative number, and so forth, the programmer must remember to follow a load instruction by a logical instruction (such as the NDA - "and the accumulator") in order to set the flags before using an instruction that is conditional in regards to a flag's status.

The description of the various types of instructions available using an '8008' CPU which follows will provide both the machine language code for the instruction given as three octal digits, and also a mnemonic name suitable for writing programs in "symbolic" type language which is usually easier than trying to remember octal codes! It may be noted that the symbolic language used is the same as that originally suggested by Intel Corporation which developed the '8008' CPU-on-a-chip. Hence users who may already be familiar with the suggested mnemonics will not have any relearning problems and those learning the mnemonics for the first time will have plenty of good company. If the programmer is not already aware of it, the use of mnemonics facilitates working with an "assembler" program when it is desired to develop relatively large and complex programs. Thus the programmer is urged to concentrate on learning the mnemonics for the instructions and not waste time memorizing the octal codes. After a program has been written using the mnemonic codes, the programmer can always use a lookup table to convert to the machine code if an assembler program is not available. It's a lot easier technique (and less subject to error) than trying to memorize

PROGRAMMING FOR THE "8008" and similar microcomputers

the 170 or so three digit combinations which make up the machine instruction code set!

The programmer must also be aware, that in this machine, some instructions require more than one word in memory. "Immediate" type commands require two consecutive words. JUMP and CALL commands require three consecutive words. The remaining types only require one word.

The first group of instructions to be presented are those that are used to load data from one CPU register to another, or from a CPU register to a word in memory, or vice-versa. This group of instructions requires just one word of memory. It is important to note that none of the instructions in this group affect the flags.

LOAD DATA FROM ONE CPU REGISTER TO ANOTHER CPU REGISTER

MNEMONIC	MACHINE CODE
LAA	300
LBA	310
.	.
.	.
LAB	301

The load register group of instructions allows the programmer to move the contents of one CPU register into another CPU register. The contents of the originating (from) register is not changed. The contents of the destination (to) register becomes the same as the originating register. Any CPU register can be loaded into any CPU register. Note that loading register A into register A is essentially a NOP (no operation) command. When using mnemonics the load symbol is the letter L followed by the "to" register and then the "from" register. The mnemonic LBA means that the contents of register A (the accumulator) is to be loaded into register B. The mnemonic LAB states that register B is to have its contents loaded into register A. It may be observed that this basic instruction has many variations. The machine language coding for this instruction is in the same format as the mnemonic code except that the letters used to represent the registers are replaced by numbers that the computer can use. Using octal code, the seven CPU registers are coded as follows:

Register A = 0
Register B = 1
Register C = 2
Register D = 3
Register E = 4
Register H = 5
Register L = 6

Also, since the machine can only utilize numbers, the octal number '3' in the most significant location of a word signifies that the computer is to perform a "load" operation. Thus, in machine coding, the instruction for loading register B with the contents of register A becomes '310' (in octal form). Or, if one wanted to get very detailed, the actual binary coding for the eight bits of information in the instruction word would be '11 001 000.' It is important to note that the load instructions do not affect any of the flags.

LOAD DATA FROM ANY CPU REGISTER TO A LOCATION IN MEMORY

LMA	370
LMB	371
LMC	372
LMD	373
LME	374
LMH	375
LML	376

This instruction is very similar to the previous group of instructions except that now the contents of a CPU register will be loaded into a specified memory location. The memory location that will receive the contents of the particular CPU register is that whose address is specified by the contents of the CPU H and L registers at the time the instruction is executed. The H CPU register specifies the HIGH portion of the address desired, and the L CPU register specifies the LOW portion of the address into which data from the selected CPU register is to be loaded. Note that there are seven different instructions in this group. Any CPU register can have its contents loaded into any location in memory. This group of instructions does not affect any of the flags.

LOAD DATA FROM A MEMORY LOCATION TO ANY CPU REGISTER

LAM	307
LBM	317
LCM	327
LDM	337
LEM	347
LHM	357
LLM	367

This group of instructions can be considered the opposite of the previous group. Now, the contents of the word in memory whose address is specified by the H (for HIGH portion of the address) and L (LOW portion of the address) registers will be loaded into the CPU register specified by the instruction. Once again, this group of instructions has no affect on the status of the flags.

LOAD IMMEDIATE DATA INTO A CPU REGISTER

LAI	006
LBI	016
LCI	026
LDI	036
LEI	046
LHI	056
LLI	066

An IMMEDIATE type of instruction requires two words in order to be completely specified. The first word is the instruction itself. The second word, or "immediately following" word, must contain the data upon which "immediate" action is taken. Thus, a load IMMEDIATE instruction in this group means that the contents of the word immediately following the instruction word is to be loaded into the specified register. For example, a typical load immediate instruction would be LAI 001. This would result in the value 001 (octal) being placed in the A register when the instruction was executed. It is important to remember that all IMMEDIATE type in-

structions MUST be followed by a data word. An instruction such as LDI by itself would result in improper operation because the computer would assume the next word contained data. If the programmer had mistakenly left out the data word, and in its place had another instruction, the computer would not realize the operator's mistake. Hence the program would be fouled-up! Note too, that the load immediate group of instructions does not affect the flags.

LOAD IMMEDIATE DATA INTO A MEMORY LOCATION

LMI	076

This instruction is essentially the same as the load immediate into the CPU register group except that now, using the contents of the H and L registers as "pointers" to the desired address in memory, the contents of the "immediately following word" will be placed in the memory location specified. This instruction does not affect the status of the flags.

The above rather large group of LOAD instructions permits the programmer to direct the computer to move data about. They are used to bring in data from memory where it can be operated on by the CPU. Or, to temporarily store intermediate results in the CPU registers during complicated and extended calculations, and of course allow data, such as results, to be placed back into memory for long term storage. Since none of them will alter the contents of the four CPU flags, these instructions can be called upon to set up data before instructions that may affect or utilize the flag's status are executed. The programmer will use instructions from this set frequently. The mnemonic names for the instructions are easy to remember as they are well ordered. The most important item to remember about the mnemonics is that the TO register is always indicated first in the mnemonic, and then the FROM register. Thus LBA equals "load TO register B FROM register A.

INCREMENT THE VALUE OF A CPU REGISTER BY ONE

INB	010
INC	020
IND	030
INE	040
INH	050
INL	060

This group of instructions allows the programmer to add one to the present value of any of the CPU registers except the accumulator. (Note carefully that the accumulator can NOT be incremented by this type of instruction. In order to add one to the accumulator a mathematical addition instruction, described later, must be used.) This instruction for incrementing the defined CPU registers is very valuable in a number of applications. For one thing, it is an easy way to have the L register successively "point" to a string of locations in memory. A feature that makes this type of instruction even more powerful is that the result of the incremented register will affect the Z, S, and P flags. (It will not change the C or "carry"flag.) Thus, after a CPU register has been incremented by this instruction, one can utilize a flag test instruction (such as the conditional JUMP and CALL instructions to be described later) to determine whether that particular register has a value of zero (Z flag), or if it is a negative number (S flag), or even parity (P flag). It is important to note that this group of instructions, and the decrement group (described in the next paragraph) are the only instructions which allow the flags to be manipulated by operations that are not concerned with the accumulator (A) register.

DECREMENT THE VALUE OF A CPU REGISTER BY ONE

DCB	011
DCC	021
DCD	031
DCE	041
DCH	051
DCL	061

The DECREMENT group of instructions is similar to the INCREMENT group except that now the value one will be subtracted from the specified CPU register. This instruction will not affect the C flag. But, it does affect the Z, S, and P flags. It should also be noted that this group, as with the increment group, does not include the accumulator register. A separate mathematical instruction must be used to subtract one from the accumulator.

ARITHMETIC INSTRUCTIONS USING THE ACCUMULATOR

The following group of instructions allow the programmer to direct the computer to perform arithmetic operations between other CPU registers and the accumulator, or between the contents of words in memory and the accumulator. All of the operations for the described addition, subtraction, and compare instructions affect the status of the flags.

ADD THE CONTENTS OF A CPU REGISTER TO THE ACCUMULATOR

ADA	200
ADB	201
ADC	202
ADD	203
ADE	204
ADH	205
ADL	206

This group of instructions will simply ADD the present contents of the accumulator register to the present value of the specified CPU register and leave the result in the accumulator. The value of the specified register is unchanged except in the case of the ADA instruction. Note that the ADA instruction essentially allows the programmer to double the value of the accumulator (which is the A register)! If the addition causes an overflow or underflow then the carry (C flag) will be affected.

ADD THE CONTENTS OF A CPU REGISTER PLUS THE VALUE OF THE CARRY FLAG TO THE ACCUMULATOR

ACA	210
ACB	211
ACC	212
ACD	213
ACE	214
ACH	215
ACL	216

This group is identical to the previous group except that the content of the carry flag is considered as an additional bit (MSB) in the specified CPU register. The combined value of the carry bit plus the contents of the specified CPU register are added to the value in the accumulator. The results are left in the accumulator. Again, with the exception of the ACA instruction, the contents of the specified CPU register are left unchanged. Again too, the carry bit (C flag) will be affected by the results of the operation.

SUBTRACT THE CONTENTS OF A CPU REGISTER FROM THE ACCUMULATOR

SUA	220
SUB	221
SUC	222
SUD	223
SUE	224
SUH	225
SUL	226

This group of instructions will cause the present value of the specified CPU register to be subtracted from the value in the accumulator. The value of the specified register is not changed except in the case of the SUA instruction. (Note that the SUA instruction is a convenient instruction with which to "clear" the accumulator.) The carry flag will be affected by the results of a SUBTRACT instruction.

SUBTRACT THE CONTENTS OF A CPU REGISTER AND THE VALUE OF THE CARRY FLAG FROM THE ACCUMULATOR

SBA	230
SBB	231
SBC	232
SBD	233
SBE	234
SBH	235
SBL	236

This group is identical to the previous group except that the content of the carry flag is considered as an additional bit (MSB) in the specified CPU register. The combined value of the carry bit plus the contents of the specified CPU register are SUBTRACTED from the value in the accumulator. The results are left in the accumulator. The carry

bit (C flag) is affected by the result of the operation. With the exception of the SBA instruction the content of the specified CPU register is left unchanged.

COMPARE THE VALUE IN THE ACCUMULATOR AGAINST THE CONTENTS OF A CPU REGISTER

CPA	270
CPB	271
CPC	272
CPD	273
CPE	274
CPH	275
CPL	276

The COMPARE group of instructions are a very powerful and somewhat unique set of instructions. They direct the computer to compare the contents of the accumulator against another register and to set the flags as a result of the comparing operation. It is essentially a subtraction operation with the value of the specified register being subtracted from the value of the accumulator except that the value of the accumulator is not actually altered by the operation. However, the flags are set in the same manner as though an actual subtraction operation had occured. Thus, by subsequently testing the status of the various flags after a COMPARE instruction has been executed, the program can determine whether the compare operation resulted in a match or non-match. In the case of a non-match, one may determine if the compared register contained a value greater or less than that in the accumulator. This would be accomplished by testing the Z flag and C flag respectively utilizing a conditional JUMP or CALL instruction (which will be described later).

ADDITION, SUBTRACTION, AND COMPARE INSTRUCTIONS THAT USE WORDS IN MEMORY AS OPERANDS

The five types of mathematical operations: ADD, ADD with CARRY, SUBTRACT, SUBTRACT with CARRY, and COMPARE, which have just been presented for the cases where they operate with the contents of CPU registers, can all be performed with words that are in memory. As with the LOAD instructions that operate with memory, the H and L registers must contain the address of the word in memory that it is desired to ADD, SUBTRACT, or COMPARE to the accumulator. The same conditions for the operations as was detailed when using the CPU registers apply. Thus, for mathematical operations with a word in memory, the following instructions are used.

ADD THE CONTENTS OF A MEMORY WORD TO THE ACCUMULATOR

ADM 207

ADD THE CONTENTS OF A MEMORY WORD PLUS THE VALUE OF THE CARRY FLAG TO THE ACCUMULATOR

ACM 217

SUBTRACT THE CONTENTS OF A MEMORY WORD FROM THE ACCUMULATOR

SUM 227

SUBTRACT THE CONTENTS OF A MEMORY WORD AND THE VALUE OF THE CARRY FLAG FROM THE ACCUMULATOR

SBM 237

COMPARE THE VALUE IN THE ACCUMULATOR AGAINST THE CONTENTS OF A MEMORY WORD

CPM 277

IMMEDIATE TYPE ADDITIONS, SUBTRACTIONS, AND COMPARE INSTRUCTIONS

The five types of mathematical operations discussed above can also be performed with the operand being the word of data immediately after the instruction. This group of instructions is similar in format to the previously described LOAD IMMEDIATE instructions. The same conditions for the mathematical operations as discussed for the operations with the CPU registers apply.

ADD IMMEDIATE

ADI 004

ADD WITH CARRY IMMEDIATE

ACI 014

SUBTRACT IMMEDIATE

SUI 024

SUBTRACT WITH CARRY IMMEDIATE

SBI 034

COMPARE IMMEDIATE

CPI 074

LOGICAL INSTRUCTIONS WITH THE ACCUMULATOR

There are several groups of instructions which allow BOOLEAN LOGIC operations to be performed between the contents of the CPU registers and the A (accumulator) register. In addition there are logic IMMEDIATE type instructions. The boolean logic operations are valuable in a number of programming applications. The instruction set allows three basic boolean operations to be performed. These are: the LOGICAL AND, the LOGICAL OR, and the EXCLUSIVE OR operations. Each type of logic operation is performed on a bit-by-bit basis between the accumulator and the CPU register or memory location specified by the instruction. A detailed explanation of each type of logic operation, and the appropriate instructions for each type is presented below. The logic instruction set is also valuable because all of them will cause the C (carry) flag to be placed in the zero condition. This is important if one is going to perform a sequence of instructions that will eventually use the status of the C flag to arrive at a decision as it allows the programmer to set the C flag to a known state at the start of the sequence. All other flags are set in accordance with the result of the logic operation. Hence, the group often has value when the programmer desires to determine the contents of a register that has just been loaded into a register. (Since the load instructions do not alter the flags.)

THE BOOLEAN 'AND' OPERATION INSTRUCTION SET

When the boolean AND instruction is executed, each bit of the accumulator will be compared with the corresponding bit in he register or memory location specified by the instruction. As each bit is compared a logic result will be placed in the accumulator for each bit comparison. The logic result is determined as follows. If both the bit in the accumulator and the bit in the register with which the operation is being performed are a logic one, then the accumulator bit will be left in the logic one condition. For all other possible combinations (A bit equals one, X bit equals zero; A bit equals zero, X bit equals one; or A bit equals zero, X bit equals zero), then the accumulator bit will be cleared to the zero state. An example will illustrate the logical AND operation.

INITIAL STATE OF THE ACCUMULATOR

1 0 1 0 1 0 1 0

CONTENTS OF OPERAND REGISTER

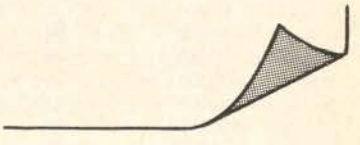
1 1 0 0 1 1 0 0

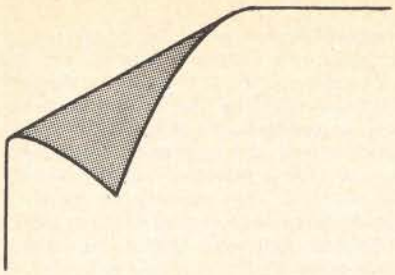
FINAL STATE OF THE ACCUMULATOR

1 0 0 0 1 0 0 0

There are seven logical AND instructions that allow any CPU register to be used as the AND operand. They are as follows.

NDA	240
NDB	241
NDC	242
NDD	243
NDE	244
NDH	245
NDL	246





The contents of the operand register is not altered by an AND logical instruction.

There is also a logical AND instruction that allows a word in memory to be used as an operand. The address of the word in memory that will be used is pointed to by the contents of the H and L CPU registers.

NDM 247

And finally there is also a logical AND IMMEDIATE type of instruction that will use the contents of the word immediately following the instruction as the operand.

NDI 044

The next group of boolean logic instructions direct the computer to perform the logical OR operation on a bit-by-bit basis with the accumulator and the contents of a CPU register or a word in memory. The logical OR operation will result in the accumulator having a bit set to a logic one if either that bit in the accumulator, or the corresponding bit in the operand register is a logic one. Since the case where both the accumulator bit and operand bit are a one also satisfies the criteria, that condition will also result in the accumulator bit being left in the one state. If neither register has a logic one in the bit position, then the accumulator bit for that position remains in the zero state. An example illustrates the results of a logical OR operation.

INITIAL STATE OF THE ACCUMULATOR

1 0 1 0 1 0 1 0

CONTENT OF THE OPERAND REGISTER

1 1 0 0 1 1 0 0

FINAL STATE OF THE ACCUMULATOR

1 1 1 0 1 1 1 0

There are seven logical OR instructions that allow any CPU register to be used as the OR operand.

ORA	260
ORB	261
ORC	262
ORD	263
ORE	264
ORH	265
ORL	266

By using the H and L registers as pointers one can also use a word in memory as an OR operand.

ORM 267

There is also the logical OR IMMEDIATE instruction.

ORI 064

As with the logical AND group of instructions, the logical OR instruction does not alter the contents of the operand register.

The last group of boolean logic instructions is a variation of the logic OR. The variation is termed the EXCLUSIVE OR logical operation. The EXCLUSIVE OR operation is similar to the OR except that when the corresponding bits in both the accumulator and the operand register are a one then the accumulator bit will be cleared to zero. Thus, the accumulator bit will be a one after the operation only if just one of the registers (accumulator register or operand register) has a one in the bit position. (Again, the operation is performed on a bit-by-bit basis.) An example provides clarification.

INITIAL STATE OF THE ACCUMULATOR

1 0 1 0 1 0 1 0

CONTENTS OF THE OPERAND REGISTER

1 1 0 0 1 1 0 0

FINAL STATE OF THE ACCUMULATOR

0 1 1 0 0 1 1 0

The seven instructions that allow the CPU registers to be used as operands are shown next.

XRA	250
XRB	251
XRC	252
XRD	253
XRE	254
XRH	255
XRL	256

The instruction that uses registers H and L as pointers to a memory location is:

XRM 257

And the EXCLUSIVE OR IMMEDIATE type instruction is:

XRI 054

As in the case of the logical OR operation, the operand register is not altered except for the special case when the XRA instruction is used. This instruction, which directs the computer to EXCLUSIVE OR the accumulator with itself, will cause the operand register, since it is the accumulator, to have its contents altered (unless it should happen to be zero at the time the instruction is executed). This is because, regardless of what value is in the accumulator, if it is EXCLUSIVE OR'ed with itself, the result will be zero! The example below illustrates the specific operation.

ORIGINAL VALUE OF ACCUMULATOR

1 0 1 0 1 0 1 0

EXCLUSIVE OR'ed WITH ITSELF

1 0 1 0 1 0 1 0

FINAL VALUE OF ACCUMULATOR

0 0 0 0 0 0 0 0

This only occurs when the logical EXCLUSIVE OR is performed on the accumulator itself. It can be shown that the results of performing the logical OR or logical AND between the accumulator and itself will result in the original accumulator value being retained.

INSTRUCTIONS FOR ROTATING THE CONTENTS OF THE ACCUMULATOR

It is often desirable to be able to shift the contents of the accumulator either right or left. In a fixed length register, a simple shift operation would result in some information being lost because what was in the MSB or LSB (depending on in which direction the shift occured) would be shifted right out of the register! Therefore, instead of just shifting the contents of a register, an operation termed ROTATING is utilized. Now, instead of just shifting a bit off the end of the register, the bit is brought around to the other end of the register. For instance, if the register is rotated to the right, the LSB (least significant bit) would be brought around to the position of the MSB (most significant bit) which would have been vacated by the shifting of its original contents to the right. Or, in the case of a shift to the left, the MSB would be brought around to the position of the LSB.

The carry bit (C flag) can be considered as an extension of the accumulator register. The instruction set for this machine allows two types of ROTATE instructions. One considers the carry bit to be part of the accumulator register for the rotate operation. The other type does not. In addition, each type of rotate can be done either to the right or to the left.

It should be noted that the rotate operations are particularly valuable when it is desired to multiply a number or divide a number. This is because shifting the contents of a register to the left effectively multiplies a binary number by a power of two. Shifting a binary number to the right provides the inverse operation.

ROTATING THE ACCUMULATOR LEFT

RLC 002

Rotating the accumulator left with the RLC instruction means the MSB of the accumulator will be brought around to the LSB position and all other bits will be shifted one position to the left. While this instruction does not shift through the carry bit, the carry bit will be set by the status of the MSB of the accumulator at the start of the ROTATE LEFT operation. (This feature allows the programmer to determine what the MSB was prior to the shifting operation by testing the C flag after the rotate instruction has been executed.

ROTATING THE ACCUMULATOR LEFT THROUGH THE CARRY BIT

RAL 022

The RAL instruction will cause the MSB of the accumulator to go into the carry bit. The initial value of the carry bit will be shifted around to the LSB of the accumulator. All other bits are shifted one position to the left.

ROTATING THE ACCUMULATOR RIGHT

RRC 012

The RRC instruction is similar to the RLC instruction except that now the LSB of the accumulator is placed in the MSB of the accumulator. All other bits are shifted one position to the right. Also, the carry bit will be set to the initial value of the LSB of the accumulator at the start of the operation.

ROTATING THE ACCUMULATOR RIGHT THROUGH THE CARRY BIT

RAR 032

Here, the LSB of the accumulator is brought around to the carry bit. The initial value of the carry bit is shifted to the MSB of the accumulator. All other bits are shifted a position to the right.

It should be noted that the C flag is the only flag that is altered by a rotate instruction. All other flags remain unchanged.

JUMP INSTRUCTIONS

The instructions discussed so far have all been "direct action" instructions. The programmer arranges a sequence of these types of instructions in memory. When the program is started the computer proceeds to execute the instructions in the order in which they are encountered. The computer automatically reads the contents of a memory location, executes the instruction it finds there, and then automatically increments a special address register called a PROGRAM COUNTER that will result in the machine reading the information contained in the next sequential memory location. However, it is often desirable to perform a series of instructions located in one section of memory, and then skip over a group of memory locations and start executing instructions in another section of memory. This action can be accomplished by a group of instructions that will cause a new address value to be placed in the PROGRAM COUNTER. This will cause the computer to go to a new section of memory and then execute instructions sequentially from the new memory location.

The JUMP instructions in this computer add considerable power to the machine's capabilities because there are a series of "conditional" JUMP instructions available. That is, the computer can be directed to test the status of a particular FLAG (C, Z, S or P). If the status of the flag is the desired one, then a JUMP will be performed. If it is not, the machine will continue to execute the next instruction in the current sequence. This capability provides a means for the computer to make "decisions" and to modify its operation as a function of the status of the various flags at the time that a program is being executed.

In a manner similar to IMMEDIATE types of instructions, the JUMP instructions require more than one word of memory. A JUMP instruction requires three words to be properly defined. (Remember that IMMEDIATE type instructions required two words.) The JUMP instruction itself is the first word. The second word must contain the LOW ADDRESS portion of the address of the word in memory that the PROGRAM COUNTER is to be set to point to, which is the new location from which the next instruction is to be fetched. The third word must contain the HIGH ADDRESS (sometimes referred to as the PAGE) of the memory address that the program counter will be set to. That is, the high order portion of the address in memory that the computer will JUMP to in order to obtain its next instruction.

THE UNCONDITIONAL JUMP INSTRUCTION

JMP 1X4

Note: The machine code 1X4 indicates that any code for the second octal digit of the machine code is valid. It is recommended as a standard practice that the code '0' be used. Thus, the typical machine code would be 104.

Remember, the JUMP instruction must be followed by two more words which contain the LOW, and then the HIGH (PAGE) portion of the address that the program is to JUMP to!

JUMP IF THE DESIGNATED FLAG IS TRUE (CONDITIONAL JUMP)

JTC	140
JTZ	150
JTS	160
JTP	170

As with the UNCONDITIONAL JUMP instruction, the CONDITIONAL JUMP instructions must be followed by two words of information. The LOW portion, then the HIGH portion, of the address that program execution is to continue from if the jump is executed. The JUMP IF TRUE group of instructions will only jump to the designated address if the condition of the appropriate flag is TRUE (logical one). Thus, the JTC instruction states that if the carry flag (C) is a logical one (TRUE) then the jump is to be executed. If it is a logical zero (FALSE) then program execution is to continue with the next instruction in the current sequence of instructions. In a similar manner the JTZ instruction states that if the ZERO FLAG is TRUE then the jump is to be performed. Otherwise the next instruction in the present sequence is executed. Likewise for the JTS and JTP instructions.

JUMP IF THE DESIGNATED FLAG IS FALSE (CONDITIONAL JUMP)

JFC	100
JFZ	110
JFS	120
JFP	130

As with all JUMP instructions these instructions must be followed by the LOW address then the HIGH address of the memory location that program execution is to continue from if the jump is executed. This group of instructions is the opposite of the jump if the flag is true group. For instance, the JFC instruction commands the computer to test the status of the carry (C) flag. If the flag is FALSE (a logic zero), then the jump is to be performed. If it is TRUE, then program execution is to continue with the next instruction in the current sequence of instructions. The same procedure holds for the JFZ, JFS and JFP instructions.

SUBROUTINE CALLING INSTRUCTIONS

Quite often when a programmer is developing computer programs the programmer will find that a particular algorithm (sequence of instructions for performing a function) can be used many times in different parts of the program. Rather than having to keep entering the same sequence of instructions at different locations in memory, which would not only consume the time of the programmer, but would also result in a lot of memory being used to perform one particular function, it is desirable to be able to be able to put an often used sequence of commands in just one location in memory. Then, whenever the particular algorithm is required by another part of the program, it would be convenient to jump to the section that contained the often used algorithm, perform the sequence of instructions, and then return back to the main part of the program. This is a standard practice in computer operations. A frequently used algorithm can be designated a SUBROUTINE. A special set of instructions allows the programmer to CALL a SUBROUTINE. In other words, specify a special type of JUMP command that will eventually allow the program to RETURN to the original "jumping" point in the program. A second type of instruction is used to terminate a SUBROUTINE. This special terminator will cause the program to revert back and pick up the next sequential in-

struction in memory that immediately follows the original CALLING instruction. A great deal of computer power is provided by the instruction set in this machine that allows one to CALL and RETURN from SUBROUTINES. This is because, in a manner similar to that provided for the CONDITIONAL JUMP instructions, there are a number of CONDITIONAL CALL and CONDITIONAL RETURN commands in the instruction set.

Like the JUMP instructions, the CALL instructions all require three words in order to be fully specified. The first word is the CALL instruction itself. The next two words must contain the LOW and HIGH portions of the starting address of the subroutine that is being "called."

When a CALL instruction is encountered by the computer, the CPU will actually save the current value of the PROGRAM COUNTER by storing it in a special PROGRAM COUNTER PUSH-DOWN STACK. This stack is capable of holding six addresses plus the current operating address. What this means is that the machine is capable of "nesting" up to seven subroutines at one time. Thus, one can have a subroutine, that in turn calls another subroutine, that in turn calls another one, up to seven levels, and the machine will still be able to return to the initial calling location. The programmer must ensure that subroutines are not nested more than seven levels otherwise the PROGRAM COUNTER PUSH-DOWN STACK will push the original calling address(es) completely out of the push-down stack. The program could then no longer automatically return to the initial calling location.

The RETURN instruction which terminates a SUBROUTINE only requires one word. When the CPU encounters a RETURN instruction it causes the PROGRAM COUNTER PUSH-DOWN STACK to "pop" up one level. This effectively causes the address saved in the stack by the calling routine to be taken as the new program counter. Hence, program execution returns to the calling location.

THE UNCONDITIONAL CALL INSTRUCTION

CAL 1X6

This instruction followed by two words containing the LOW and then the HIGH order of the starting address of the SUBROUTINE that is to be executed is an UNCONDITIONAL CALL. The subroutine will be executed regardless of the status of the FLAGS. The next sequential address after the CAL instruction is saved in the PROGRAM COUNTER PUSH-DOWN STACK.

THE UNCONDITIONAL RETURN INSTRUCTION

RET 0X7

This instruction directs the CPU to unconditionally "pop" the program counter push-down stack UP one level. Program execution will continue from the address saved by the subroutine calling instruction.

CALL A SUBROUTINE IF THE DESIGNATED FLAG IS TRUE

CTC	142
CTZ	152
CTS	162
CTP	172

In a manner similar to the conditional JUMP IF TRUE instructions, these instructions (which must all be followed by the LOW and HIGH portions of the called subroutine's starting address) will only perform the "call" if the designated flag is in the TRUE (logical one) state. If the designated flag is FALSE then the CALL instruction is ignored. Program execution then continues with the next sequential instruction.

RETURN FROM A SUBROUTINE IF THE DESIGNATED FLAG IS TRUE

RTC	043
RTZ	053
RTS	063
RTP	073

These one word instructions will cause a SUBROUTINE to be TERMINATED only if the designated flag is in the logical one (TRUE) state.

CALL A SUBROUTINE IF THE DESIGNATED FLAG IS FALSE

CFC	102
CFZ	112
CFS	122
CFP	132

These instructions are the opposit of the previous group of calling commands. The subroutine is called only if the designated flag is in the FALSE (logical zero) condition. Remember, these instructions must be followed by two words which contain the LOW and HIGH part of the starting address of the SUBROUTINE that is to be executed if the designated flag is FALSE. If the flag is TRUE, the subroutine will not be called and program operation will continue with the next instruction in the current sequence.

RETURN FROM A SUBROUTINE IF THE DESIGNATED FLAG IS FALSE

RFC	003
RFZ	013
RFS	023
RFP	033

These one word instructions will terminate a subroutine ("pop" the program counter stack UP one level) if the designated flag is FALSE. Otherwise, the instruction is ignored and program operation is continued with the next instruction in the subroutine.

THE SPECIAL RESTART SUBROUTINE CALL INSTRUCTIONS

There is a special purpose instruction available that effectively serves as a one word SUBROUTINE CALL. (Remember that it normally requires three words to specify a subroutine call.) This special instruction allows the programmer to call a subroutine that starts at any one of eight specially designated memory locations. The eight special memory locations are at locations: 000, 010, 020, 030, 040, 050, 060 and 070 on page zero. There are eight variations of the machine code for the RESTART instruction. One for each of the above addresses. Thus, the one word instruction can serve to CALL a SUBROUTINE at the specified starting location (instead of having two additional words to specify the starting address of a subroutine). It is often convenient to utilize a RESTART command as a quick CALL to an often used subroutine. Or, as an easy way to call short "starting" subroutines for large programs. Hence, the name for the type of instruction. The eight RESTART instructions, in their mnemonic and machine code forms, along with the starting address associated with each one is listed below.

RST 0	005	00 000
RST 1	015	00 010
RST 2	025	00 020
RST 3	035	00 030
RST 4	045	00 040
RST 5	055	00 050
RST 6	065	00 060
RST 7	075	00 070

INPUT INSTRUCTIONS

In order to receive information from an external device the computer must utilize a group of special signal lines. The typical '8008' computer is designed to handle up to eight groups (each group having eight signal lines) of INPUT signals. A group of signals is accepted at the computer by what is referred to as an INPUT PORT. The computer controls the operation of the INPUT PORTS. Under program control, the computer can be directed to obtain the information that is on a group of lines coming in to any INPUT PORT. When this is done the information will be transferred to the accumulator. Various types of external equipment, such as an electronic keyboard or measuring instruments, can be connected to the INPUT PORTS. The INPUT PORTS are typically referred to as having numbers from '0' to '7.' The typical mnemonics and machine codes for INPUT instructions are shown next.

INP 0	101
INP 1	103
.	.
INP 6	115
INP 7	117

It may be interesting to note that the machine codes for input ports increase by a factor of two for each port. Note too, that while the mnemonic for an input instruction has two parts, the machine code only requires one word in memory. It is also important to realize that while an input instruction brings data into the accumulator it does not affect the status of any of the CPU flags!

OUTPUT INSTRUCTIONS

In order to output information to an external device the computer utilizes another group of signal lines which are referred to as OUTPUT PORTS. A Typical '8008' system may be equipped to service up to twenty-four OUTPUT PORTS. (Each OUTPUT PORT actually consists of eight signal lines.) An OUTPUT instruction causes the contents of the accumulator to be transferred to the signal lines of the designated OUTPUT PORT. The output ports are normally designated by octal numbers in the range 10 to 37. The list below shows the typical mnemonics used to specify an OUTPUT PORT along with the associated machine code. (It may be interesting to note again that the machine code increases by a factor of two for each port.)

OUT 10	121
OUT 11	123
.	.
OUT 21	141
.	.
OUT 36	175
OUT 37	177

An OUTPUT instruction only requires one machine code word (even though the mnemonic is typically specified in two parts). OUTPUT PORTS are connected to external devices that one desires to have the computer transmit information to, such as a CRT display, or machinery that is to be placed under computer control.

THE HALT INSTRUCTION

There is one more instruction in the '8008' instruction set. This instruction directs the CPU to stop all operations and to remain in that state until an INTERRUPT signal is received. In a typical '8008' system an INTERRUPT signal may be generated by an operator pressing a switch or by an external piece of equipment sending an electronic signal to the CPU. This instruction is normally used when the programmer desires to terminate a program or when it is desired to have the computer wait for an operator or external device to perform some action. There are three machine codes that may be used for the HALT command.

HLT	000
HLT	001
HLT	377

The HALT instruction does not affect the status of the CPU flags.

INFORMATION ON INSTRUCTION EXECUTION TIMES

When programming for "real-time" applications it is important to know how much time each type of instruction requires to be executed. With this information the programmer can develop "timing loops" or determine with substantual accuracy how much time it will take to perform a particular series of instructions. This information is especially valuable when dealing with programs that control the operations of external devices which might require events to occur at specific times.

The following table provides the nominal instruction execution time for each category of instruction used in an '8008' system. The precise time needed for each instruction depends on how close the master clock has been set to a nominal value of 500 kilohertz. The table shows the number of cycle states required by the type of instruction followed by the nominal time required to perform the entire instruction. Since each state executes in four microseconds, the total time required to perform the instruction as shown in the table was obtained by multiplying the number of states by four microseconds. By knowing the number of states required for each instruction the programmer can often rearrange an algorithm or substitute different types of instructions to provide programs that have events occuring at precisely timed intervals.

INSTRUCTION EXECUTION TIME TABLE

LOAD DATA FROM A CPU REGISTER TO ANOTHER CPU REGISTER	5	20 Us
LOAD DATA FROM A CPU REGISTER TO A LOCATION IN MEMORY	7	28
LOAD DATA FROM MEMORY TO A CPU REGISTER	8	32
LOAD IMMEDIATE DATA INTO A CPU REGISTER	8	32
LOAD IMMEDIATE DATA INTO A LOCATION IN MEMORY	9	36
INCREMENT OR DECREMENT A CPU REGISTER	5	20
ARITHMETIC/COMPARE BETWEEN ACCUMULATOR & A CPU REGISTER	5	20
ARITH/COMPARE BETWEEN ACCUMULATOR & A WORD IN MEMORY	8	32
IMMEDIATE.ARITHMETIC AND COMPARE	8	32
BOOLEAN OPS BETWEEN ACCUMULATOR AND CPU REGISTERS	5	20
BOOLEAN OPS WITH ACCUMULATOR & A WORD IN MEMORY	8	32
IMMEDIATE BOOLEAN OPERATIONS	8	20
ROTATE THE ACCUMULATOR	5	20
JUMP AND CALL COMMANDS (UNCONDITIONAL)	11	44
JUMP/CALLS WHEN CONDITION NOT SATISFIED (CONDITIONAL)	9	36
JUMP/CALLS WHEN CONDITION SATISFIED (CONDITIONAL)	11	44
RETURN (UNCONDITIONAL)	5	20
RETURN WHEN CONDITION NOT SATISFIED (CONDITIONAL)	3	12
RETURN WHEN CONDITION SATISFIED (CONDITIONAL)	5	20
RESTART COMMAND	5	20
OUTPUT COMMAND	6	24
INPUT COMMAND	8	32
HALT COMMAND	4	16

Chapters 2 and 3 of **MACHINE LANGUAGE PROGRAMMING FOR THE "8008" (and similar microcomputers)** will appear in BYTE's August and September issues, respectively.■

BYTE Reprint

Chapter 2

MACHINE LANGUAGE PROGRAMMING FOR THE "8008" and similar microcomputers

Reprinted from **MACHINE LANGUAGE PROGRAMMING FOR THE "8008" (and similar microcomputers).**

Author: Nat Wadsworth

INITIAL STEPS FOR DEVELOPING PROGRAMS

The first task that should be done prior to starting to write the individual instructions for a computer program is to decide exactly what it is that the computer is to perform and to write the goal(s) down on paper! This statement might seem unnecessary to some because it is such an obvious one. It is stated because the majority of people learning to develop programs will realize its significance when they discover, halfway through the writing of a large machine language program, that they left out a vital step. Such an error can typically result in the programmer having to start back at the beginning and rewrite the entire program. The practice of writing down just what tasks a particular program is to perform and the steps in which they are to be done, will save a lot of work in the long run. The written description should be as complete and detailed as necessary to ensure that exactly each step of the program will be clear when actually writing the program in machine language. It is generally wise for the novice programmer to take pains to be quite detailed in the initial description.

The act of actually writing down the proposed operation of the program desired serves several valuable purposes. First, it forces one to carefully review what is planned. In doing so, it often vividly reveals flaws in original mental ideas. Secondly, it serves as a guide and a check list as the machine language program is developed. Remember, it will often take a number of hours to write a fair sized program. These hours might be spread over several days or weeks. In this period of time the human mind can easily forget original intentions and plans if the human memory is not refreshed by written notes. A program that is not kept carefully organized as it is developed can become a real mess. This is especially so if one keeps forgetting key concepts or has to constantly add in forgotten routines. The time wasted by such sloppy procedures can be avoided if proper work habits are developed from the beginning.

Once one has written a description of the general task(s) to be performed, and has ascertained that there are no flaws to the overall concepts or ideas, it is a good idea to draw up a set of FLOW CHARTS for the proposed program. FLOW CHARTS are detailed written and symbolic descriptive diagrams of the flow of operations that are to occur as the program is executed. They also show the interrelationships between different portions of a program.

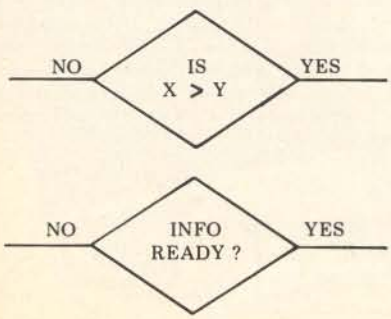
Over the years a variety of symbols and methods have been developed for creating flow charts. All of the varieties have the same basic purpose and most of the differences are the result of individuals pushing their own preferences. Most people can do admirably well using just a few basic symbols to denote fundamental types of operations in a computer program. The small group to be presented here will enable most microcomputer programmers to develop flow charts rapidly, with little confusion, and without having to learn a host of special symbols.

A CIRCLE may be used as a general purpose symbol to specify an entry or exit point in a routine or subroutine. Information may be printed inside the circle. This information might denote where the routine is coming from or going to (such as the page number and location on a page for a program that requires several sheets of paper to be flow charted). It might contain transfer

information. Or, it could denote the starting and stopping points within a program. Some typical examples of the CIRCLE symbol are illustrated next.

A square or rectangel may be used to denote a general or specific operation. The type of operation may be described inside the box such as illustrated in the following examples.

A diamond form may be used to symbolize a decision or branching point in a program. The determining factor(s) for the decision or branching operation may be indicated inside the symbol. The two side points of the diamond are used to illustrate the path taken when a decision has been made. The diamond symbol is illustrated next.

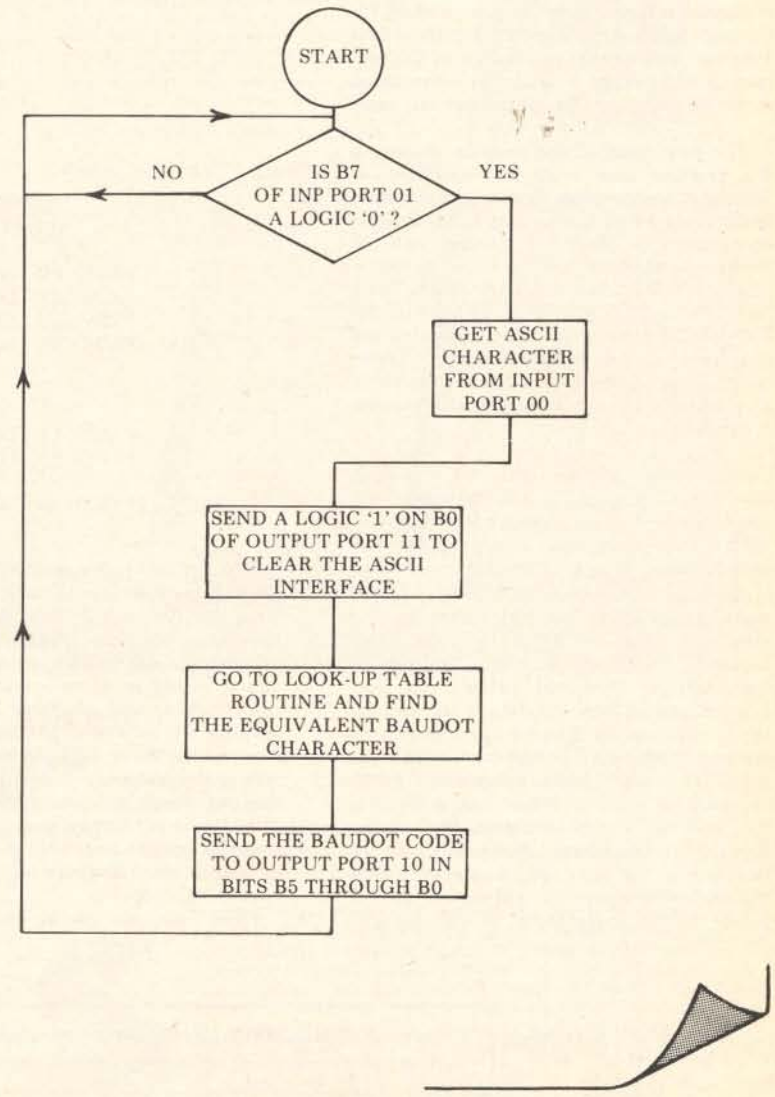


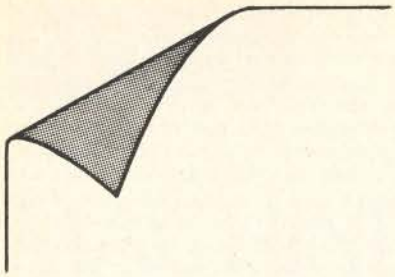
Lines with arrows may be used to interconnect the three types of symbols presented. In this way, the symbols may be connected to form readily understood FLOW CHARTS of operations that are to occur in a program and to show how various operations relate to each other. Flow charts are extremely valuable references when developing programs as well as when one wants to update or expand a program and needs to quickly review the operation of the program of specific interest.

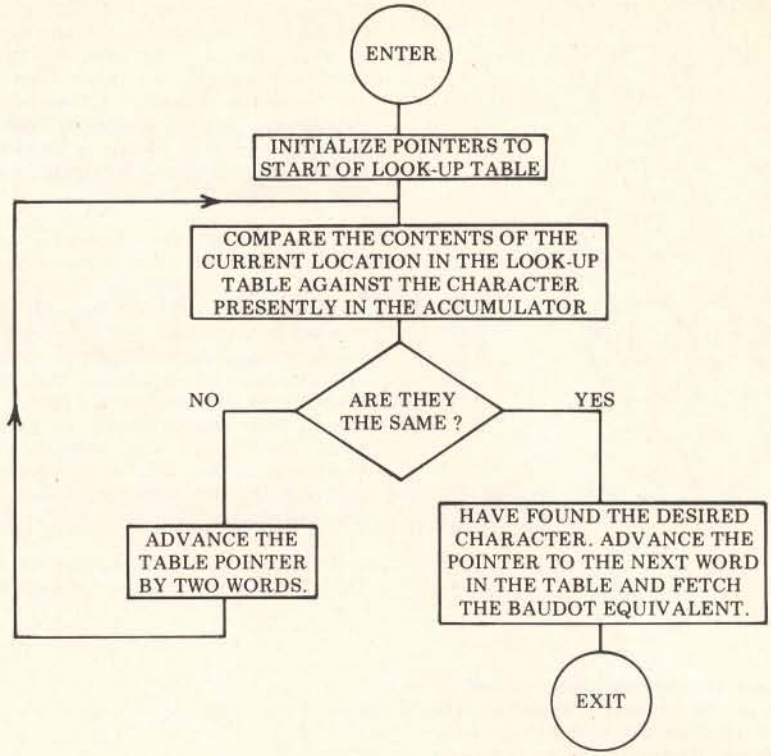
An example of a flow chart for a relatively simple program will be shown next. The program illustrated by the flow chart is to accept characters from an ASCII encoded electric typewriter and send out the equivalent character to a BAUDOT coded device. In this illustration it is assumed that the I/O interfaces to the machines are parallel interfaces (versus the possibility of being bit-serial interfaces). Thus, complex timing operations do not have to be discussed in the example. A written description of the example program could be stated as follows.

The computer is to monitor bit B7 of INPUT PORT 01, which is the control port for an interface to an ASCII encoded electric typewriter. Whenever bit B7 on INPUT PORT 01 goes low (logic '0') it indicates a new character is waiting in parallel format from the typewriter at INPUT PORT 00. The computer is to immediately obtain the character that is waiting at INPUT PORT 00 and as soon as it has obtained the data it is to send a logic '1' (high) signal to bit B0 of OUTPUT PORT 11 to signal the ASCII interface that the character has been accepted by the computer. (The receipt of this signal by the ASCII interface will then cause the ASCII interface to restore the control signal on bit B7 of INPUT PORT 01 to a high (logic '1') condition.)

Whenever a character has been received from the ASCII typewriter on INPUT PORT 00, the computer is to compare the character just received against an ASCII to BAUDOT look-up table which is stored in the computer's memory until it finds a match. It will then obtain the equivalent BAUDOT character from the conversion table. It will then send the BAUDOT code for the character in bit positions B5 through B0 of OUTPUT PORT 10. Bit B5 will serve to indicate to the BAUDOT interface whether







the code in bits B4 through B0 is to be processed by the BAUDOT device when it is in the LETTERS or FIGURES mode. It is assumed that the character rate (but not necessarily the baud rate) is the same for both machines so that the example may be simplified by eliminating the requirement for character buffering or stacking in the memory of the computer. However, in practical applications such capability might be required. The feature could be added to the program. However, for this case, as soon as the BAUDOT code has been transmitted (in parallel format) to the BAUDOT device, the computer will simply go back to waiting for the next character to come in from the ASCII machine. The written description of the program just presented is succinctly summarized in the flow chart shown on the previous page!

The flow chart of the program shown on the previous page could be considered an outline of the program. Portions of that flow chart could be expanded into more detailed flow charts to present a detailed view of special operations. For instance, the rectangle labeled GO TO LOOK-UP TABLE ROUTINE AND FIND THE EQUIVALENT BAUDOT CHARACTER really refers to a portion of the program that consists of a number of operations. Those operations could be described in a separate flow chart such as the one just presented.

The reader can see that the expanded flow chart illustrates the operation of the table look-up routine portion of the program. With a little study one can discern that the look-up table consist of an area in memory that has an ASCII encoded character in one word, followed in the next word by the same character in BAUDOT code. This sequence continues for all the possible characters as illustrated below. The flow chart illustrates how the data in the look-up table is scanned by skipping over every other memory location (which contains the BAUDOT codes) until the proper ASCII character is located. When that is located, the routine simply extracts the proper BAUDOT code from the next memory locaction in the table. The flow chart makes the sequence easier to understand than a purely verbal explanation of the routine.

ADDRESS	MEMORY CONTENTS
PAGE: XX LOC: Z	ASCII code for letter A
PAGE: XX LOC: Z+1	BAUDOT code for letter A
PAGE: XX LOC: Z+2	ASCII code for letter B
PAGE: XX LOC: Z+3	BAUDOT code for letter B
. .	.
. .	.
. .	.
PAGE: XX LOC: Z+2(N-1)	ASCII code for N'th letter
PAGE: XX LOC: Z+2(N-1)+1	BAUDOT code for N'th letter

ILLUSTRATION OF LOOK-UP TABLE ORGANIZATION FOR THE EXAMPLE PROGRAM

It is strongly recommended that beginning programmers develop the habit of first writing down the function(s) of the desired program they intend to create. Next, one should draw up flow charts as detailed as one feels is necessary to clearly show the operation of the program that is to be developed. A novice programmer will be wise to prepare quite detailed flow charts. More experienced programmers may prefer to leave out details of operations that they thoroughly understand. Flow charts should serve as ready references when the programmer goes on to actually develop the step-by-step machine language instruction sequences for the computer.

Flow charts are also an excellent method for communicating programming concepts to fellow computer technologists. Remember that general flow charts do not have to be machine specific!) Learning how to prepare and read flow charts is an important (yet easy) skill for all computer programmers to acquire. It can also be fun and a highly creative process. Using the technique, one may review the overall operation of a program under development and gain new insights into where to interconnect routines, where common loops exist (which can save valuable memory room if they are subroutined), and find other ways in which to enhance a program's capabilities.

Chapter 3 of **MACHINE LANGUAGE PROGRAMMING FOR THE "8008" (and similar microcomputers)** will appear in the September BYTE.

Chapter 3

BYTE Reprint

MACHINE LANGUAGE PROGRAMMING FOR THE "8008" and similar microcomputers

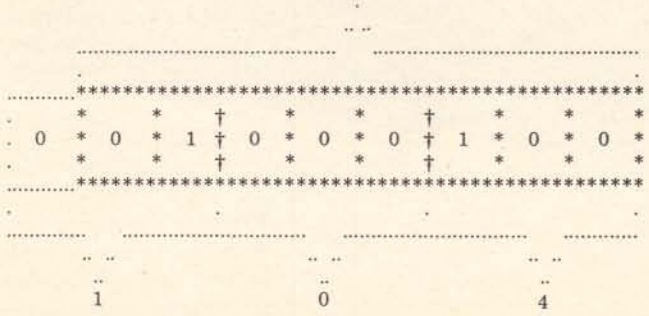
FUNDAMENTAL PROGRAMMING SKILLS

Before one can effectively develop machine language programs for a computer, one must be thoroughly familiar with the instruction set for the machine. It is assumed for the remainder of this manual that the reader has studied the detailed information for the instruction set of the 8008 CPU which was provided in the first chapter. The programmer should become intimately familiar with the mnemonics (pronounced kneemonics) for each type of instruction. Mnemonics are easily remembered symbolic representations of machine language instructions. They are far easier to work with than the actual numeric codes used by the computer when the programmer is developing a program. While the programmer will develop programs and think in terms of the mnemonics, the programmer must eventually convert the mnemonics to the machine codes used by the computer. This, however, is almost purely a look-up procedure. In fact, as will be seen shortly, this task can actually be performed by the computer through the use of an ASSEMBLER program.

Machine language programmers should also be familiar with manipulating numbers in binary and octal form. It is assumed that readers are familiar with representing numbers as binary values. However, there may be a few readers who are not used to the convention of representing binary numbers by their octal equivalents. The technique is quite simple. It consists merely of grouping binary digits into groups of three and representing their value as an octal number. The octal numbering system only uses the digits 0 through 7. This is exactly the range that a group of three binary digits can represent. The octal numbering system makes it a lot easier to manipulate binary numbers. For instance, most people find it considerably more convenient to remember a three digit octal number such as 104 than the binary equivalent 01000100. An octal number is easily expanded to a binary number by simply placing the octal value in binary form using three binary digits.

The information in an eight bit binary register can be readily converted to an octal number by grouping the bits into groups of three starting with the least significant bits. The two most significant bits in the register which form the last group will only be able to represent the octal numbers 0 to 3. The diagram below illustrates the convention.

EIGHT CELL REGISTER



CONVERTING AN 8 BIT REGISTER FROM BINARY TO OCTAL NUMBERS

Note in the diagram how an imaginary additional binary digit with a value of zero was assigned to the left of the most significant bit so that the octal convention for the two most significant bits could be maintained.

A table illustrating the relationship between the binary and octal systems is provided for reference below.

BINARY PATTERN	REPRESENTATIVE OCTAL NO.
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	4
1 0 1	5
1 1 0	6
1 1 1	7

A person who desires to develop machine language programs for computers should become familiar with standard conventions used when dealing with closed registers (groups of binary cells of fixed length such as a memory word or CPU register). One very simple point to remember is that when a group of cells in a register is in the all ones condition:

1 1 1 1 1 1 1 1

and a count of 1 is added to the register, the register goes to the value:

0 0 0 0 0 0 0 0

Or, if a count of: 10 (binary) was added to a register that contained all ones, the new value in the register would be as shown:

```
  1 1 1 1 1 1 1 1
+ 0 0 0 0 0 0 1 0
-----------------
  0 0 0 0 0 0 0 1
```

Similarly, going the opposite way, if one subtracts a number such as 100 (binary) from a

Reprinted from **MACHINE LANGUAGE PROGRAMMING FOR THE "8008" (and similar microcomputers).**

register that contains some lesser value, such as 010 (binary), the register would contain the result shown below:

```
00 000 010
00 000 100
-----------
11 111 110
```

It may be noted that if one uses all the bits in a fixed length register, one may represent mathematical values with an absolute magnitude from zero to the quantity two to the Nth power, minus one (0 to (2**N - 1)) where N is the number of bits in the register. If all the bits in a register are used to represent the magnitude of a number, and it is also desired to represent the magnitude as being either positive or negative in sign, then some additional means must be available to record the sign of the magnitude. Generally, this would require using another register or memory location solely for the purpose of keeping track of the sign of a number.

In many applications it is desirable to establish a convention that will allow one to manipulate positive and negative numbers without having to use an additional register to maintain the sign of a number. One way this may be done is to simply assign the most significant bit in a register to be a sign indicator. The remaining bits represent the magnitude of the number regardless of whether it is positive or negative. When this is done, the magnitude range for an N cell register becomes 0 to (2**(N-1))-1 rather than 0 to (2**N) - 1. The convention normally used is that if the most significant bit in the register is a one then the number represented by the remaining bits is negative in sign. If the MSB is zero, then the remaining bits specify the magnitude of a positive number. This convention allows computer programmers to manipulate mathematical quantities in a fashion that makes it easy for the computer to keep track of the sign of a number. Some examples of binary numbers in an eight bit register are shown next.

BINARY REPRESENTATION	OCTAL	DECIMAL
00 001 000	010	+ 8
10 001 000	210	- 8
01 111 111	177	+127
11 111 111	377	- 127
00 000 001	001	+ 1
10 000 001	201	- 1

While the signed bit convention allows the sign of a number to be stored in the same register (or word) as the magnitude, simply using the signed bit convention alone can still be a somewhat clumsy method to use in a computer. This is because of the method in which a computer mathematically adds the contents of two binary registers in the accumulator. Suppose, for example, that a computer was to add together positive and negative numbers that were stored in registers in the signed bit format.

```
      00 001 000  (+ 8 decimal)
PLUS  10 001 000  (- 8 decimal)
-----------------
EQUAL 10 010 000  (This is not 0!)
```

The result of the operation illustrated would not be what the programmer intended! In order for the operation to be performed correctly, it is necessary to establish a method for processing the negative number called the two's complement convention. In the two's complement convention, a negative number is represented by complementing what the value for a positive number would be (complementing is the process of replacing bits that are '0' with a '1,' and those that are '1' with a 0) and then adding the value one (1) to the complemented value. As an example, the number minus eight (-8) decimal would be derived from the number plus eight (+8) by the following operations.

```
00 001 000      (Original + 8)

11 110 111      (Complemented)
00 000 001      (now add +1)
-----------
11 111 000      (2's complement
                 form of - 8)
```

Some examples of numbers expressed in two's complement notation with the signed bit convention are shown below.

BINARY REPRESENTATION	OCTAL	DECIMAL
00 001 000	010	+ 8
11 111 000	370	- 8
01 111 111	177	+127
10 000 001	201	- 127
00 000 001	001	+ 1
11 111 111	377	- 1
00 000 000	000	+ 0
10 000 000	200	- 128

Note that when using the two's complement method, one may still use the convention of having the MSB in the register establish the sign. If the MSB = 1, as in the above illustration, the number is assumed to be negative. Since the number is in the two's complement form, the computer can readily add a positive and a negative number and come up with a result that is readily interpreted. Look!

```
      00 001 000 (+ 8 decimal)
ADD   11 111 000 (- 8 dec as 2's comp)
-----------------------
      00 000 000 (Correct answer = 0)
```

Another established convention in handling numbers with a computer is to assume that '0' is a positive value. Because of this convention, the magnitude of the largest negative number that can be represented in a fixed length register is one more than that possible for a positive number.

The various means of storing and manipulating the signs of numbers as just discussed have advantages and drawbacks, and the method used depends on the specific application. However, for most user's, the two's complement signed bit convention will be the most convenient, most often used, method. The prospective machine language programmer should make sure that the convention is well understood.

Another area that the machine language programmer must have a thorough knowledge of is the conversion of numbers between the decimal numbering system that most people work with on a daily basis, and the binary and octal numbering system utilized by computer technologists. Programmers working with microcomputers will generally find the octal numbering system most convenient. Because the conversion from octal to binary is simply a matter of grouping binary bits into groups of three as discussed at the start of this chapter, it is easier to remember octal codes than long strings of binary digits. However, most people are used to thinking in decimal terms, which the computer does not use at the machine language level. Thus, it is necessary for programmers to be able to convert back and forth between the various numbering systems as programs are developed.

The conversion process that is generally the most troublesome for people to learn is from decimal to binary, or decimal to octal (and vice-versa)! It is usually a bit easier for people to learn to convert from decimal to octal, and then use the simple octal to binary expansion technique, than to convert directly from decimal to binary. The easier method will be presented here. It is assumed that the reader is already familiar with going from octal to binary (and vice-versa). Only the conversions between decimal and octal (and the reverse) will be presented at this point.

A decimal number may be converted to its octal equivalent by the following technique:

Divide the decimal number by 8. Record the remainder (note that is the REMAINDER!!) as the least significant digit of the octal number being derived. Take the quotient just obtained and use it as the new dividend. Divide the new dividend by 8. The remainder from this operation becomes

the next significant digit of the octal number. The quotient is again used as the new dividend. The process is continued until the quotient becomes '0.' The number obtained from placing all the remainders (from each division) in increasing significant order (first remainder as the least significant digit, last remainder as the most significant digit) is the octal number equivalent of the original decimal. The process is illustrated below for clarity.

The octal equivalent of 1234 decimal is:

ORIGINAL NUMBER	1234	/	8	=	154	2
LAST QUOTIENT BECOMES NEW DIVIDEND	154	/	8	=	19	2 .
LAST QUOTIENT BECOMES NEW DIVIDEND	19	/	8	=	2	3 . .
LAST QUOTIENT BECOMES NEW DIVIDEND	2	/	8	=	-	2 . . .
Thus the octal equivalent of 1234 decimal is:						2 3 2 2

The above method is quite easy and straightforward. Since a majority of the time the user will be interested in conversions of decimal numbers less than 255 (the maximum decimal number that can be expressed in an eight bit register) only a few divisions are necessary:

The octal equivalent of 255 decimal is:

					QUOTIENT	REMAINDER
ORIGINAL NUMBER	255	/	8	=	31	7
LAST QUOTIENT BECOMES NEW DIVIDEND	31	/	8	=	3	7
LAST QUOTIENT BECOMES NEW DIVIDEND	3	/	8	=	-	3
Thus the octal equivalent of 255 is:						3 7 7

For numbers less than 63 decimal (and such numbers are used frequently to set counters in loop routines) the above method reduces to one division with the remainder being the LSD and the quotient the MSD. This is a feat most programmers have little difficulty doing in their head!

The octal equivalent of 63 decimal is:

ORIGINAL NUMBER	63	/	8	=	7	7
LAST QUOTIENT BECOMES NEW DIVIDEND	7	/	8	=	-	7
Thus the octal equivalent of 63 is:						7 7

Going from octal to decimal is quite easy too. The process consists of simply multiplying each octal digit by the number 8 raised to its positional (weighted) power, and then adding up the total of each product for all the octal digits:

2 3 2 2 Octal =

. 2	X	(8*0)	=	(2 X 1)	=	2
. . . 2	X	(8*1)	=	(2X8)	=	1 6
. . 3	X	(8*2)	=	(3 X 64)	=	1 9 2
2	X	(8*3)	=	(2 X 512)	=	1 0 2 4
Thus the decimal equivalend of 2322 Octal is :						1 2 3 4

Besides the basic mathematical skills involved with using octal and binary numbers, there are some practical bookkeeping considerations that machine language programmers must learn to deal with as they develop programs. These bookkeeping matters have to do with memory usage and allocation.

As the reader who has read chapter one in this manual knows, each type of instruction used in the 8008 CPU requires one, two, or three words of memory. As a general rule, simple register to register or register to memory commands require but one memory word. Immediate type commands require two memory locations (the instruction code followed immediately by the data or operand). Jump or call instructions require three words of memory storage. One word for the instruction code and two more words for the address of the location specified by the instruction. The fact that different types of instructions require different amounts of memory is important to the programmer.

As programmers write a program it is often necessary for them to keep tabs on how many words of memory the actual operating portion of the program will require (in addition to controlling the areas in memory that will be used for data storage). One reason for maintaining a count of the number of memory words a program requires is simply to ensure that the program will fit into the available memory space.

Often a program that is a little too long to be stored in an available amount of memory when first developed can be rewritten, after some thought, to fit in the available space. Generally, the trade-off between writing compact programs versus not-so-compact routines is simply the programmer's development time. Hastily constructed programs tend to require more memory storage area because the programmer does not take the time to consider memory conserving instruction combinations.

However, even if one is not concerned about conserving the amount of memory used by a particular program, one still often needs to know how much space a group of instructions will consume in memory. This is so that one can tell where another program might be placed without interfering with a previous program.

For these reasons, programmers often find it advantageous to develop the habit of writing down the number of memory words utilized by each instruction as they write the mnemonic sequences for a routine. Additionally, it is often desirable to maintain a column showing the total number of words required for storage of a routine. An example of a work sheet with this practice being followed is illustrated here:

MEMORY WORDS THIS INSTR.	TOTAL WORDS THIS ROUTINE	MNEMONICS	COMMENTS
2	2	LAI 000	Place 000 in accumulator
2	4	LHI 001	Set Register H to 1
2	6	LLI 150	And Regis L to 150
1	7	ADM	Add the contents of memory
1	8	INL	Locations 150 & 151 on page 1
1	9	ADM	Adding second number to first
1	10	RET	End of subroutine

In the example the total number of words used in column was kept using decimal numbers. Many programmers prefer to maintain this column using octal numbers because of the direct correlation between the total number of words used, and the actual memory addresses used by the 8008.

The example just presented can be used to introduce another consideration during program development. That is memory allocation. One must distinguish between program storage areas in memory, and areas used to hold data that is operated on by the program. Note that the sample subroutine was designed to have the computer add the contents of memory locations 150 and 151 on page 01. Thus, those two locations must be reserved for data. One must ensure that those specific memory locations are not inadvertantly used for some other purpose. In a typical program, one may have many locations in memory assigned for holding or manipulating data. It is important that one maintain some sort of system of recording where one plans to store blocks of data and where various operating routines will reside as a program is developed. This can be readily accomplished by setting up and using memory usage maps (often commonly referred to as core maps). An example of a memory usage map being started for the subroutine just discussed is shown.

MEMORY USAGE MAP

PG	LOC	MACHINE CODE			LABELS	MNEMONICS	COMMENTS
01	000				ADD,		Add no's @ 150 & 151
01	010						
01	020						
01	030						
01	040						
01	050						
01	060						
01	070						
01	100						
01	110						
01	120						
01	130						
01	140						
01	150						Number storage
01	151						Number storage
01	152						
01	153						
01	154						
01	155						
01	156						
01	157						
01	160						
01	170						
01	200						

PROGRAM DEVELOPMENT WORK SHEET

PG	LOC	MACHINE CODE			LABELS	MNEMONICS	COMMENTS
01	000	006	000		ADD,	LAI 000	Set ACC = 000
01	002	056	001			LHI 001	Set pntr PG = 1
01	004	066	150			LLI 150	Set pntr LOC = 150
01	006	207				ADM	Add 1'st no. to ACC
01	007	060				INL	Adv pntr to 2'nd no.
01	010	207				ADM	Add 2'nd no. to 1'st
01	011	007				RET	Exit subroutine

The same type of form may also be used as a program development sheet as shown here . One may observe that the form provides for memory addresses, the actual octal values of the machine codes, labels and mnemonics used by the programmer, and additional information.

Memory usage maps are extremely valuable for keeping large programs organized as they are developed, or for displaying the locations of a variety of different programs that one might desire to have residing in memory at the same time. It is suggested that the person intending to do even a moderate amount of machine language programming make up a supply of such forms (using a ditto or mimeograph machine) to have on hand.

There are some important factors about machine language programming that should be pointed out as they have considerable impact on the total efficiency and speed at which one can develop such programs and get them operating correctly. The factors relate to one simple fact. People developing machine language programs (especially beginners) are very prone to making programming mistakes! Regardless of how carefully one proceeds, it always seems that any fair sized program needs to be revised before a properly operating program is achieved. The impact that changes in a program have on the development (or redevelopment) effort vary according to where in the program such changes must be made. The reason for the seriousness of the problem is because program changes generally result in the addresses of the instructions in memory being altered. Remember, if an instruction is added, or deleted, then all the remaining instructions in the routine being altered must be moved to different locations! This can have multiplying effects if the instructions that are moved are referred to by other routines (such as call and jump commands) because then the addresses referred to by those types of commands must be altered too! To illustrate the situation, a change will be made to the sample program presented several pages ago. Suppose it was decided that the subroutine should place the result of the addition calculation in a word in memory before exiting the subroutine, instead of simply having the result in the accumulator. The original program, for example, could have been residing in the locations shown on the program development work sheet on the previous page. Changing the program would result in it occupying the following memory locations:

PAGE	LOC	MEMORY CONTENTS	MNEMONICS	COMMENTS
01	000	006	LAI 000	Place 000 in accumulator
01	001	000		
01	002	056	LHI 001	Set Reg H to 1
01	003	001		
01	004	066	LLI 150	Set Reg L to 150
01	005	150		
01	006	207	ADM	Add contents of memory
01	007	060	INL	Locations 150 & 151
01	010	207	ADM	Add 2nd to 1st
01	011	066	LLI 160	Set Reg L to 160
** 01	012	160		
** 01	013	370	LMA	Save answer @ 160
** 01	014	007	RET	End of subroutine

The ** locations denote the additional memory locations required by the modified subroutine. If the programmer had already developed a routine that resided in locations 012, 013, or 014, the change would require that it be moved!

If one was using a program development work sheet, one would have had to erase the original RET instruction at the end of the routine and then written in the two new commands, and added the RET instruction at the end. The effects would not be too devestating since the change was inserted at the end of the subroutine. But, suppose a similar change was necessary at the start of a subroutine that had 50 instructions in it? The programmer would have to do a lot of erasing!

The effects of changes in program source listings was recognized early as a problem in developing programs. Because of this people developed programs called EDITORS that would enable the computer to assist people in the task of creating and manipulating source listings for programs. An EDITOR is a program that will allow a person to use a computer as a text buffer. Source listings may be entered from a keyboard or other input device and stored in the computer's memory. Information that is placed in the text buffer is kept in an organized fashion, usually by lines of text. An Editor program generally has a variety of commands available to the operator to allow the information stored in the text buffer to be manipulated. For instance, lines of information in the text buffer may be added, deleted, moved about or inserted before other lines, and so forth. Naturally, the information in the buffer can be displayed to the operator on an output device such as a cathode ray tube (CRT) or electromechanical printing mechanism. Using this type of program, a programmer can rapidly create a source listing and modify it as necessary. When a permanent copy is desired, the contents of the text buffer may be punched on paper tape or written on a magnetic tape cassette. It turns out that the copy placed on paper tape or a cassette can often be further processed by another program to be discussed shortly which is termed an ASSEMBLER program. However, an important reason for making a copy of the text buffer on paper tape or magnetic cassette tape is because if it is ever necessary to make changes to the source listing, then the old listing can be quickly reloaded back into the computer. Changes may then be rapidly made using the Editor program, and a new clean listing obtained in a fraction of the time that might be required to erase and rewrite a large number of lines using pencil and paper.

Relatively small programs can be developed using manual methods. That is, by writing the source listings with pencil and paper. But, anyone that is planning on doing extensive program development work should obtain an Editor program in order to substantually increase their overall program development efficiency. Besides, an Editor program can be put to a lot of good uses besides just making up source listings! Such as enabling one to edit correspondence or prepare written documents that are nice and neat in a fraction of the time required by conventional methods.

Changes in source listings naturally result in changes to the machine codes (which the mnemonics simply symbolize). Even more important, the addresses associated with instructions often must be changed due to additions or deletions of words of machine code. For instance, in the example routine being used in this section, memory address PAGE 01 LOCATION 011 originally contained the code for a RET (RETURN) instruction which is 007. When the subroutine was changed by adding several more instructions (so the answer could be stored in a memory location), the RET instruction was shifted down to the address PAGE 01 LOCATION 014. The address where it formerly resided was changed to hold the code for the first part of the LLI 160 instruction which is 066. Had changes been made earlier in the routine, then many more memory locations would need to be assigned different machine codes. However, the changes caused by adding on to the sample program previously discussed are not as far reaching as the one presented on the following page. There the changes result in the addresses of subroutines referred to by other routines being changed, so that it is then necessary to go back and modify the machine codes in all of the routines that refer to the subroutine that was changed!

PAGE	LOC	MEMORY CONTENTS	MNEMONICS		COMMENTS
00	000	026	OVER,	LCI 100	Load reg C with 100
00	001	100			
00	002	106		CAL NEWONE	Call a new subroutine
00	003	013			
00	004	000			
00	005	106		CAL LOAD	And then another
00	006	023			
00	007	000			
00	010	104		JMP OVER	Jump back & repeat
00	011	000			
00	012	000			
00	013	056	NEWONE,	LHI 000	Load reg H with zeroes
00	014	000			
00	015	066		LLI 200	And L with 200
00	016	200			
00	017	317		LBM	Fetch mem contents to B
00	020	010		INB	Increment the value in B
00	021	371		LMB	Place B back into memory
00	022	007		RET	End of subroutine

PAGE	LOC	MEMORY CONTENTS	MNEMONICS		COMMENTS
00	023	056	LOAD,	LHI 003	Set H to PG 03
00	024	003			
00	025	361		LLB	Place register B into L
00	026	370		LMA	Place ACC into memory
00	027	021		DCC	Decrement value in reg C
00	030	013		RFZ	Return if C is not zero
00	031	000		HLT	Halt when C = zero

Suppose it was decided to insert a single word instruction right after the LCI 100 command in the above program. The new program would appear as shown next.

PAGE	LOC	MEMORY CONTENTS	MNEMONICS		COMMENTS
00	000	026	OVER,	LCI 100	Load reg C with 100
00	001	100			
00	002	250		XRA	Clear the accumulator
* 00	003	106		CAL NEWONE	Call a new subroutine
* 00	004	** 014			
* 00	005	000			
* 00	006	106		CAL LOAD	And then another
* 00	007	** 024			
* 00	010	000			
* 00	011	104		JMP OVER	Jump back and repeat
* 00	012	000			
* 00	013	000			
* 00	014	056	NEWONE,	LHI 000	Load Reg H with zeroes
* 00	015	000			
* 00	016	066		LLI 200	And L with 200
* 00	017	200			
* 00	020	317		LBM	Fetch mem contents to B
* 00	021	010		INB	Increment the value in B
* 00	022	371		LMB	Place B back into memory
* 00	023	007		RET	Exit subroutine
* 00	024	056	LOAD,	LHI 003	Set H to PAGE 03
* 00	025	003			
* 00	026	361		LLB	Place reg B into L
* 00	027	370		LMA	Place ACC into memory
* 00	030	021		DCC	Decrement value in reg C
* 00	031	013		RFZ	Return if C is not zero
* 00	032	000		HLT	Halt when C is zero

Note in the illustration how not only the addresses of all the instructions beyond location 002 (denoted by the *) change, but even more important, that parts of the instructions themselves (the address portion of the CAL instructions, denoted by the **) must now be altered. The essential point being made here is that if the starting address of a routine or subroutine that is referred to by any other part of the program is changed, then each and every reference to that routine must be located and the address portion corrected! This can be an extremely formidable, time consuming, tedious, and down right frustrating task if all the references must be found and corrected by manual means in a large program!

Early computer technologists soon became disgusted with making such program corrections by hand methods after learning that it was almost impossible to develop large programs without making a few errors. They went to work on finding a method to ease the task of making such corrections and came up with a type of program called an ASSEMBLER that could utilize the computer itself to perform such exacting tasks. ASSEMBLER programs are types of programs that are able to process source listings when they have been written in mnemonic (symbolic) form and translate them into the OBJECT code (actual machine language code) that is utilized directly by the computer. An ASSEMBLER also keeps track of assigning the proper addresses to references to routines and subroutines. This is accomplished through a process initiated by the programmer assigning LABELS to routines in the source listing. One may now see that the combination of an Editor and an Assembler program can greatly ease the task of developing machine language programs over that of the purely manual method. The use of such programs is almost mandatory when programs become large because the manual method becomes highly unwieldy. A primary reason that an Editor and Assembler are so useful is because if a mistake is made in the program, one can use the relatively quick method of utilizing the Editor program to revise the source listing. Then, one may use the Assembler program to reprocess the corrected source listing and produce a new version of the machine code assigned to new addresses if appropriate.

For quite small programs, say less than 100 instructions, the use of Editor and Assembler programs are not mandatory. In fact, even if one uses these aids for small programs, one should know how to manually convert mnemonic listings to object code. This is because it may occasionally be desirable to make minor program changes (patches) without having to go through the process of using an Editor and Assembler. This is particularly true when one is DEBUGGING large programs and wants to ascertain whether a minor correction will correct a problem. The process of converting from a mnemonic listing to actual machine code is not difficult in concept. Many readers will have discerned the process from the examples already provided. However, for any who are in doubt, the process will be explained for the sake of clarity.

Suppose a person desired to produce a small program that would set the contents of all the words in PAGE 01 of memory to 000. The programmer would first develop the algorithm and write it down as a mnemonic (source) listing. Such an algorithm might appear as follows.

	MNEMONIC	COMMENTS
	LHI 001	Set the high address register to PAGE 01.
	LLI 000	Set the low address register to the first location on the page assigned by reg. H.
AGAIN,	LMI 000	Load the contents of the memory location specified by registers H & L to 000.
	INL	Advance register L to the next memory location (but do not change the page).
	JFZ AGAIN	If the value of register L is not 000 after it has been incremented then JUMP back to the part of the program denoted by the label AGAIN and repeat the process.
	HLT	If the value of register L is 000, then have the computer stop as the program is done!

To convert the source listing to machine (object) code the programmer must first decide where the program is to reside in memory. In this particular case it would certainly not be wise to place the program anywhere on PAGE 01 as the program would self-destruct! The program could safely be placed anywhere else. For the sake of demonstration it will be assumed that it is to reside on PAGE 02 starting at LOCATION 100. To convert the source listing to machine code the programmer would simply make a list of the addresses to be occupied by the program. Then the programmer would simply look up the machine code corresponding to the mnemonic for each instruction and place this number next to the address in which it will reside. (The machine code for each mnemonic used by the '8008' CPU is provided in Chapter ONE of this manual.)

Since some instructions are location dependent in that they require the actual address of referenced routines, it is often necessary to assign the machine code in two processes. The first process consist of assigning the machine codes to specific memory addresses wherever possible. When the machine code requires an address that has not yet been determined, the memory location is left blank. The second process consists of going back and filling in any blanks once the addresses of referenced routines have been determined. In the example being used for illustration, only one process is required because the address specified by the label AGAIN is defined before the label (address) is referenced by the JFZ instruction. The sample program when converted to machine language code would appear as shown next.

ORIGINAL MNEMONIC	MEMORY ADDRESS	MEMORY CONTENTS	COMMENTS
LHI 001	02 100	056	Machine code for LHI mnemonic
	02 101	001	Immediate part of LHI mnemonic
LLI 000	02 102	066	Machine code for LLI mnemonic
	02 103	000	Immediate part of LLI mnemonic
AGAIN, LMI 000	02 104	076	Machine code for LMI mnemonic Note that the label AGAIN now defines an address of LOCATION 104 on PAGE 02
	02 105	000	Immediate part of LMI mnemonic
INL	02 106	060	Increment low address here
JFZ AGAIN	02 107	110	Machine code for JFZ mnemonic
	02 110	104	Low address portion of the CONDITIONAL JUMP instruction as defined by label AGAIN above
	02 111	002	PAGE address portion of the CONDITIONAL JUMP instruction defined by label AGAIN
HLT	02 112	377	Alternately, the code 000 or 001 could have been used here as the machine code for a HALT command

---

Once the program has been put in machine language form the actual machine code may be placed in the assigned locations in memory. The programmer may then proceed to verify the algorithm's validity. For small programs such as the example just illustrated the machine code can simply be loaded into the correct memory locations using manual methods typically provided on microcomputer systems. Such small programs can then be easily checked out by stepping through the program one instruction at a time.

If the program is relatively large then a special loader program which is typically provided with an ASSEMBLER program could be used to load in the machine code.

Checking out and DEBUGGING large programs can sometimes be difficult if a few simple rules are not followed. A good rule of thumb is to first test out each subroutine independently. One may choose to STEP through a subroutine, or else to place HALT instructions at the end of each subroutine. Then one may verify that data was manipulated properly by a particular subroutine before going on to the next section in a program. The use of strategically located HALT instructions in a program initially being tried out is an important technique for the programmer to remember. When a HALT is encountered the user may check the contents of memory locations and examine the contents of CPU registers to determine if they contain the proper values at that point in the program. (Using the manual operator controls and indicator lamps typically provided with microcomputer development systems.) If all is well at a check point then the programmer may replace the HALT instruction with the actual instruction for that point. One may then continue checking the operation of the program after making certain that any registers that were altered by the examination procedure (typically registers H and L in an '8008' system) have been reset to the desired values if they will effect operation of the program as it continues!

It is often helpful to use a utility program known as a MEMORY DUMP program to check the contents of memory locations when testing a new program. A memory dump program is a small utility program that will allow the contents of areas in memory to be displayed on an output device. Naturally, the memory dump program must reside in an area of memory outside that being used by the program being checked. By using this type of program the operator may readily verify the contents of memory locations before and after specific operations occur to see if their contents are as expected. A memory dump program is also a valuable aid in determining whether a program has been properly loaded or that a portion of a program is still intact after a program under test has gone errant.

One will find that having flow charts and memory maps at hand during the DEBUGGING process is also very helpful. They serve as a refresher on where routines are supposed to be in memory and what the routines are supposed to be doing.

If minor corrections are necessary or desired, then one may often make program corrections, or PATCHES as they are commonly referred to by software people, to see if the corrections believed appropriate will work as planned. An easy way to make a PATCH to a program is to replace a CALL or JUMP instruction with a CALL to a new subroutine that contains the desired corrections (plus the original CALL or JUMP instruction if necessary). If a CALL or JUMP instruction is not available in the vicinity of the area where a correction must be made then one can replace three words of instructions with a CALL patch provided that one is very careful not to split up a multi-word instruction. If this cannot be avoided, then the remaining portion of a split-up multi-word instruction must be replaced with a NO-OPERATION instruction such as a LAA command (in an '8008' system). One must also make certain that the instructions displaced by the inserted CALL instruction are placed in the patching subroutine (provided that they are not being removed purposely). An example of several patches being made to the small example program previously discussed will be illustrated next.

Suppose, in the example just presented, that the operator decided not to clear (set to 000) all the words in PAGE 01 of memory, but rather to only clear the locations 000 to 177 (octal) on the page. The program could be modified by replacing the JFZ AGAIN instruction which started at LOCATION 107 on PAGE 02 with the command CAL 000 003 (CALL the subroutine starting at LOCATION 000 on PAGE 03 which will be the PATCH). Now at LOCATION 000 on PAGE 03 one could put:

MNEMONIC	MEMORY ADDRESS	MEMORY CONTENTS	COMMENTS
LAI 200	03 000	006	Put value 200 into the ACC
	03 001	200	Note value of 200 used because contents of register L has been incremented
CPL	03 002	276	Compare contents of the ACC with the contents of register L
JFZ AGAIN	03 003	110	If accumulator and L do not
	03 004	104	match then continue with the
	03 005	002	original program
RET	03 006	007	End of PATCH subroutine

Suppose instead of filling every word on PAGE 01 with zeroes the programmer decided to fill every other other word? A patch could be made by replacing the LMI 000 command at LOCATION 106 on PAGE 02 and again inserting a CAL 000 003 command to a patch subroutine that might appear as illustrated below.

MNEMONIC	MEMORY ADDRESS	MEMORY CONTENTS	COMMENTS
LMI 000	03 000	076	Keep the LMI instruction
	03 001	000	as part of the PATCH
INL	03 002	060	Keep original increment L
INL	03 003	060	And add another increment L to skip every other word
RET	03 004	007	Exit from PATCH subroutine

Finally, to illustrate a patch that splits a multi-word command, consider a hypothetical case where the programmer decided that prior to doing the clearing routine, it would be important to save the contents of register H before setting it to PAGE 01. If a three word CALL command is placed starting at LOCATION 100 on PAGE 02 in the original routine to serve as a PATCH, it may be observed that the second half of the LLI 000 instruction would cause a problem when the program returned from the patch. (The value of 000 at LOCATION 103 on PAGE 02 in the example program would be interpreted as a HLT command by the computer when it returned from the patch subroutine.) In order to avoid this problem the programmer could place a LAA (effectively a NO-OPERATION command) at LOCATION 103 on PAGE 02 after placing the patch command CAL 000 003 instruction beginning at LOCATION 100 on PAGE 02. The actual patch subroutine might appear as shown below.

MNEMONIC	MEMORY ADDRESS	MEMORY CONTENTS	COMMENTS
LEH	03 000	345	Save register H in register E
LHI 001	03 001	056	Now set register H to point
	03 002	001	to PAGE 01
LLI 000	03 003	066	And set the low address
	03 004	000	pointer to LOCATION 000
RET	03 005	007	End of PATCH subroutine

In the balance of this manual numerous techniques for developing machine language programs will be presented and discussed. Many of the examples used will be presented as subroutines that the reader may use when developing customized programs. It is important for the new programmer to learn to think of programs in terms of routines or subroutines and then learn to combine subroutines into larger programs. This practice makes it easier for the programmer to initially develop programs. It is generally much easier to create small algorithms and then combine them, in the form of subroutines, into larger programs. Remember, subroutines are sequences of instructions that can be CALLED by other parts of a program. They are terminated by RETURN or CONDITIONAL RETURN commands. It is also wise when developing programs to leave some room in memory between subroutines so that patches can be inserted or routines lengthened without having to rearrange the contents of a large amount of memory. Finally, while speaking of subroutines, it will be pointed out that the user would be wise to keep a note book of subroutines that the individual develops in order to build up a reference library of pertinent routines. It takes time to think up and check out algorithms. It is very easy to forget just how one had solved a particular problem six months after one initially accomplished the task. Save your accrued efforts. The more routines you have to utilize, the more valuable your machine becomes. The power of the machine is all determined by WHAT YOU PUT IN ITS MEMORY!

1. First, the programmer should clearly define and write down on paper exactly what the program is to accomplish.

2. Next, flow charts to aid in the complex task of writing the mnemonic (source) listings are prepared. They should be as detailed as necessary for the programmer's level of experience and ability.

3. Memory maps should be used to distribute and keep track of program storage areas and data manipulating regions in available memory.

4. Using the flow charts and memory maps as guides, the actual source listings of the algorithms are written using the symbolic representations of the instructions. An Editor program is frequently used to good advantage at this point.

5. The mnemonic source listings are converted into the actual machine language numerical codes assigned to specific addresses in memory. An Assembler program makes this task quite easy and should be used for large programs.

6. The prepared machine code is loaded into the appropriate addresses in the computer's memory and operation of the program is verified. Often the initial check out is done using the STEP mode of operation, or by exercising individual subroutines. The judicial use of inserted HALT instructions at key locations will often be of value during the initial testing phase.

7. If the program is not performing as intended then problem areas must be isolated. Program PATCHES may be utilized to make minor corrections. If serious problems are found it may be necessary to return to step no. 3, or step no. 1! ■